

# DESCRIPTION OF THE CHORD PROTOCOL USING ASMS FORMALISM

BOJAN MARINKOVIĆ, PAOLA GLAVAN, AND ZORAN OGNJANOVIĆ

**ABSTRACT.** This paper describes the overlay protocol Chord using the formalism of Abstract State Machines. The formalization concerns Chord actions that maintain ring topology and manipulate distributed keys. We define a class of runs and prove the correctness of our formalization with respect to it.

**Keywords:** Peer-to-peer, Chord, DHT-based Overlay Networks, Abstract State Machines, Formalization.

## 1. INTRODUCTION

A decentralized Peer-to-Peer system (P2P) [23] involves many peers (nodes) which execute the same software, participate in the system having equal rights and might join or leave the system continuously. In such a framework processes are dynamically distributed to peers, with no centralized control. P2P systems have no inherent bottlenecks and can potentially scale very well. Moreover, since there are no dedicated nodes critical for systems' functioning, those systems are resilient to failures, attacks, etc. The main applications of P2P-systems involve: file sharing, redundant storage, real-time media streaming, etc.

P2P systems are frequently implemented in a form of overlay networks [24], a structure that is totally independent of the underlying network that is actually connecting devices. Overlay network represents a logical look on organization of the resources. Some of the overlay networks are realized in the form of Distributed Hash Tables (DHT) that provide a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating peer can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the peers, in such a way that any change in the set of participants causes a minimal amount of disruption. It allows a DHT to scale to extremely large number of peers and to handle continual node arrivals, departures, and failures. The Chord protocol [20–22] is one of the first, simplest and most popular DHTs. The paper [20] which introduces Chord has been recently awarded the SIGCOMM 2011 Test-of-Time Award.

Our aim is to describe Chord using Abstract State Machine (ASM) [11] and to prove the correctness of the formalization, which was motivated by the obvious fact that errors in concurrent systems are difficult to reproduce and find merely by program testing. There are at least two reasons for using ASMs. First, ASMs are versatile machines which are able to simulate arbitrary algorithms in a direct and essentially coding-free way. Here the term algorithm is taken in a broad sense including programming languages, architectures, distributed and real-time protocols, etc. The simulator is not supposed to implement the algorithm on a lower abstraction level; the simulation should be performed on the natural abstraction level of

the algorithm. Second, a vast literature on ASMs shows how to model closely and faithfully real complex systems and how to use models in order to verify their properties (see for example [5], Bakery algorithm [4], Rail road crossing problem [13], Kerberos algorithm [3], Java formalization [7], [8], a special issue devoted to the method [6], etc). The ASM-code for Chord presented in this paper has been written following one of the best implementations [10] of the high level C++-like pseudo code from [22]. Actually, the main difference between the implementation [10] and our specification concerns improving efficiency of detection of a failed successor proposed in [25] which we have added to the code.

Recently, several non-relational database systems (NRDBMS) have been developed [9, 16] that are usually based on the Chord like technology. To analyze their behavior, it might be useful to characterize situations when correctness of the underlying protocol holds. Following that idea, we have formulated several deterministic conditions that guarantee correctness of Chord, and proved the corresponding statements. This is in contrast to the approach from [17, 20–22] where a probabilistic analysis is proposed, and correctness holds with "high probability".

The main objectives of Chord are maintaining the ring topology as nodes concurrently join and leave a network, mapping keys onto nodes and distributed data handling. The formalism of ASM enables us to precisely describe a class of possible runs - so called regular runs - of the protocol, and to prove correctness of the main operations with respect to it. Moreover, several examples of runs, given in Example 5.1, that violate the constraints for the regular runs illustrate how correctness can be broken in those cases.

We are aware of only a few attempts to formally verify behavior of DHTs and particularly Chord [1, 2, 15, 17, 25–27]. We consider them below and compare with our approach.

The rest of the paper is organized in the following way. Section 2 introduces the notion of ASM. Chord is informally described in Section 3. Our ASM formalization of Chord is given in Section 4, where we present the general program and a high level description of the rules executed by every peer. In Section 5 the class of regular runs is defined, and statements and the corresponding proofs of the correctness of the formalization are given. Section 6 presents discussion and comparison with related works. We conclude in Section 7. In Appendix A, we provide a detailed specification of the rules involved in our formalization of Chord.

## 2. ABSTRACT STATE MACHINE

We assume that the reader is familiar with the semantics of the Abstract State Machine defined in [5, 11, 12], and we quote here only the essential definitions.

A Gurevich's Abstract State Machine  $\mathcal{A}$  is defined by a program *Prog* - consisting of a finite number of *transition rules*, at most countable set of states and initial states.  $\mathcal{A}$  models the operational behavior of a real dynamic system  $\mathcal{S}$  in terms of evolution of states.

A state  $S$  is a first-order structure over a fixed signature (which is also the signature of  $\mathcal{A}$ ), representing the instantaneous configuration of  $\mathcal{S}$ . The value of a term  $t$  at  $\mathcal{S}$  is denoted by  $[t]_S$ . The basic transition rule is the following function update

$$f(t_1, \dots, t_n) := t$$

where  $f$  is an arbitrary  $n$ -ary function and  $t_1, \dots, t_n, t$  are first-order terms. To fire this rule in a state  $S$  evaluate all terms  $t_1, \dots, t_n, t$  at  $S$  and update the function  $f$  to  $[t]_S$  on parameters  $[t_1]_S, \dots, [t_n]_S$ . This produces another state  $S'$  which differs from  $S$  only in the new interpretation of the function  $f$  (since states represent memory, function update represents change in the content of one memory location).

Additionally, we have the following transition rules.

The *conditional constructor* produces “guarded” transition rules of the form:

```
if  $g$  then
   $R_1$ 
else
   $R_2$ 
endif
```

where  $g$  is a ground term (the guard of the rule) and  $R_1, R_2$  are transition rules. To fire that new rule in a state  $S$  evaluate the guard; if it is true, then execute  $R_1$ , otherwise execute  $R_2$ . The **else** part may be omitted.

The *seq constructor* produces transition rules of the form:

```
seq
   $R_1$ 
  ...
   $R_n$ 
endseq
```

to apply  $R_1, \dots, R_n$  sequentially. Note that the *seq*-constructor is originally defined without the **endseq**-line, but we add it to improve readability.

The *par constructor* produces transition rules of the form:

```
par
   $R_1$ 
  ...
   $R_n$ 
endpar
```

to apply  $R_1, \dots, R_n$  simultaneously, when possible, otherwise do nothing.

If  $U$  is a universe name,  $v$  is a variable,  $g(v)$  is a Boolean term and  $R$  is a rule then the following expression (*choose constructor*) is a rule with the main existential variable  $v$  that ranges over  $U$  and body  $R$ :

```
choose  $v$  in  $U$  satisfying  $g(v)$ 
   $R$ 
endchoose
```

If there is an element  $a \in U$  such that condition  $g(a)$  is true, fire rule  $R$  (with  $a$  substituted for  $v$ ), otherwise do nothing.

To express the simultaneous execution of a rule  $R$  for each  $x$  satisfying a given condition  $\varphi$  (*forall constructor*):

```
forall  $x$  with  $\varphi$  do
   $R$ 
endforall
```

A run (or computation) of  $\mathcal{A}$  is a finite or infinite sequence  $S_0; S_1; S_2; \dots$  where  $S_0$  is an initial state and every  $S_{i+1}$  is obtained from  $S_i$  executing a transition rule.

In general runs may be affected by the environment. Environment manifests itself via so-called external functions. Every external function can be understood as a (dynamic) oracle. The ASM provides the arguments and the oracle gives the result.

In a distributed Gurevich's Abstract State Machine  $\mathcal{A}$  multiple autonomous agents cooperatively model a concurrent computation of  $\mathcal{S}$ . Each agent  $a$  executes its own single-agent program  $Prog(a)$  as specified by the module associated with  $a$  by the function  $Mod$ . More precisely, an agent  $a$  has a partial view  $View(a; S)$  of a given global state  $S$  as defined by its sub-vocabulary  $Fun(a)$  (i.e. the function names occurring in  $Prog(a)$ ) and it can make a move at  $S$  by firing  $Prog(a)$  at  $View(a; S)$  and changing  $S$  accordingly. The underlying semantic model ensures that the order in which the agents of  $\mathcal{A}$  perform their actions is always such that no conflicts between the update sets computed for distinct agents can arise. The global program  $Prog$  is the union of all single-agent programs. Nullary function  $Me$ , that allows an agent to identify itself among other agents, is interpreted as  $a$  for each agent  $a$ , and does not belong to  $Fun(a)$  for any agent  $a$ . It cannot be the subject of an update instruction and is used to parameterize the agent's specific functions. A sequential run of a distributed Gurevich's Abstract State machine  $\mathcal{A}$  is a (finite or infinite) sequence  $S_0; S_1; \dots; S_n; \dots$  of states of  $\mathcal{A}$ , where  $S_0$  is an initial state and every  $S_{n+1}$  is obtained from  $S_n$  by executing a move of an agent. The partially ordered run, defined in [11], is the most general definition of runs for a distributed ASM. In order to prove properties on a partially ordered run, the attention may be restricted to a linearization of it, which is, in turn, a sequential run (see [11] for more explanations). In the rest of the paper we consider only *regular runs* in which a state is global and moves of agents are atomic.

### 3. (INFORMAL) DESCRIPTION OF CHORD

We assume that a number of nodes running the Chord protocol form a ring-shaped network. The main operation supported by Chord is:

- mapping the given key onto a node using consistent hashing.

The consistent hashing [14] provides load-balancing, i.e., every node receives roughly the same number of keys, and only a few keys are required to be moved when nodes join and leave the network. Chord networks are overlay systems. Thus, each node in a network (with  $N$ -nodes) needs "routing" information about only a few other nodes ( $O(\log N)$ ), and resolves all lookups via  $O(\log N)$  messages to other nodes. When the network is not stable, i.e., the corresponding "routing" information is out of date since nodes join and leave arbitrarily, the performance degrades. However, a recent development [25] has shown that Chord's stabilization algorithm (with minor modifications) maintains good lookup performance despite continuous failure and joining of nodes.

Identifiers are assigned to nodes and keys by the consistent hash function. The identifier for a node or a key,  $hash(node)$  or  $hash(key)$ , is produced by hashing IP of the node, or the value of the key. The length of identifiers (for example  $m$  bits) must guarantee that the probability that two objects of the same type are assigned same identifiers is negligible. Identifiers are ordered in an identifier circle modulo  $2^m$ . Then, the key  $k$  is assigned to the node such that  $hash(node) = hash(key)$ . If

such a node does not exist, the key is assigned to the first node in the circle whose identifier is greater than  $hash(key)$ .

Every node possesses information on its current successor and predecessor nodes in the identifier circle. To accelerate the lookup procedure, a node also maintains routing information in the form of the so-called *Finger Table* with up to  $m$  entries. The  $i^{th}$  entry in the table at the node  $n$  contains the identifier of the first node  $s$  that succeeds  $n$  by at least  $2^{i-1}$  in the identifier circle, i.e.,  $s = successor(n + 2^{i-1})$ , where  $1 \leq i \leq m$  (and all arithmetic is performed modulo  $2^m$ ). The stabilization procedure implemented by Chord must guarantee that each node's successor pointer and finger table are up to date. The procedure runs periodically in the background at each node. To increase robustness, each Chord node can create a successor list of size  $r$ , containing the node's first  $r$  successors.

Beside the mapping of keys onto the set of nodes, the only other operations realized by Chord are:

- adding/removing of a node to/from a network.

When a node  $n$  joins an existing network, certain keys previously assigned to  $n$ 's successor now become assigned to  $n$ . When node  $n$  leaves the network regularly, it notifies its predecessor and successor and reassigns all of its keys to the successor.

#### 4. ASM FORMALIZATION OF CHORD

**4.1. Basic Notions.** Let  $L$ ,  $M$  and  $K$  be three fixed positive integers, and  $N = 2^M$ . We will consider the following disjoint universes:

- the set  $Peer = \{p_1, \dots, p_L\}$  of all peers that might participate in the considered Chord network,
- the set  $Key = \{k_1, \dots, k_K\}$  of identifiers of objects that might be stored in the considered Chord network, and the set  $Value = \{v_1, \dots, v_K\}$  of the values of those  $K$  objects,
- the set  $Chord = \{0, 1, \dots, N - 1\}$  denoting at most  $N$  peers that are involved in the network in a particular moment,
- the sets  $Join = \{join, skip\}$  and  $Action = \{put, get, fair\_leave, unfair\_leave, skip\}$  which represent the actions of the peers,
- the sets  $Mode = \{not\_connected, connected\}$ ,  $Mode\_join = \{undef, wait\_for\_successor, wait\_for\_keys, finished\}$ ,  $Mode\_leave = \{undef, wait\_for\_successor, proceed\_to\_finish\}$ ,  $Mode\_stabilize = \{undef, wait\_for\_successor, wait\_for\_predecessor, wait\_for\_keys, wait\_for\_successor\_keys, finished\}$ ,  $Mode\_get = \{undef, wait\_for\_key, wait\_for\_value\}$ ,  $Mode\_put = \{undef, wait\_for\_response\}$ ,  $Mode\_fingers = \{undef, wait\_for\_response\}$  which represent the states of the peers,
- the set  $CommunicationType = \{get, find\_successor, send\_successor, send\_predecessor, send\_keys, set\_predecessor, request\_and\_remove\_keyvalue, successor\_found, value\_found, get\_predecessor, received\_keyvalue, received\_predecessor\}$  of special nullary functions denoting type of communication,
- the set  $ContentType = \{Chord \times Value\}^* \cup \{Chord \times Chord\} \cup Chord \cup Value$  which represents all possible types of messages content, and
- the set  $Message = CommunicationType \times ContentType$  which represents messages that are exchanged by the nodes. A messages is defined by its type and content.

Note that:

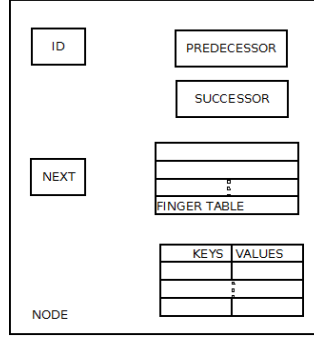


FIGURE 1. Structure of Chord node

- it might be that  $L > N$  ( $K > N$ ), i.e., that there are more peers (objects to be stored in the network) than nodes, but it can never be  $N > L$ , and
- without any loss of generality we assume that the numbers of keys and values are the same; if there are more values than keys, all values mapped to the same key might be organized in a list.

In the sequel, we will use the following (standard) list-manipulation functions:

- *list constructor*,
- *add* a new element to the list,
- *remove* an element from the list, and
- *listitem* returns the  $i^{th}$  element of a list,

and the strict and the total orders of  $\mathbb{N}$  (denoted by  $<$ , and  $\leq$ ). Also, we will use  $a \oplus_N b$  to denote  $(a + b) \bmod N$ .

Any peer, active in the network will be called a *node*. We assume that a *node* (Fig. 1) is represented by its identifier  $id$  in the network, information on its *predecessor* and *successor*, a *finger table*, a pointer (*next*) to an element in the finger table which will be updated in the current stabilization cycle, and a *list* of  $\langle key, value \rangle$  pairs of the records for which the node is responsible for.

More formally, we introduce the following functions:

- $id : Peer \rightarrow Chord \cup \{undef\}$
- $successor : Chord \rightarrow Chord$ ,
- $predecessor : Chord \rightarrow Chord$ ,
- $finger : Chord \rightarrow Chord^*$ ,
- $next : Chord \rightarrow \{1, \dots, M\}$ , and
- $keyvalue : Chord \rightarrow (Chord \times Value)^*$ ,

where  $Chord^*$  is the set that contains lists of nodes' identifiers, and  $(Chord \times Value)^*$  is the set of lists containing pairs  $\langle hash(key), value \rangle$ . Each  $finger(x)$  has  $M$  entries ordered respect to the ring ordering.

In other words, a peer  $p$ , which is a node, is represented by the tuple:

- $\langle id(p), successor(id(p)), predecessor(id(p)), finger(id(p)), next(id(p)), keyvalue(id(p)) \rangle$ .

Table 1 shows all the other functions that will be used in the formal description of the protocol, but that do not directly change the representations of nodes. We assume that the five functions in Table 1 (*hash*, *ping*, *known\_nodes*, *key\_value* and *keys*) are external.

Function	Description
<i>hash</i>	Maps the sets of peers and keys to $Chord \cup \{undef\}$
<i>ping</i>	Tests whether a node is reachable
<i>member_of</i>	Checks whether a node is between two nodes in <i>Chord</i>
<i>communication</i>	Realizes communication requests
<i>mode</i>	Determines <i>Peer_agent</i> state
<i>mode_join</i>	Determines <i>Peer_agent</i> state during join operation
<i>mode_leave</i>	Determines <i>Peer_agent</i> state during fair leave operation
<i>mode_stabilize</i>	Determines <i>Peer_agent</i> state during stabilize operation
<i>mode_fingers</i>	Determines <i>Peer_agent</i> state during update of finger table member
<i>mode_put</i>	Determines <i>Peer_agent</i> state during put operation
<i>mode_get</i>	Determines <i>Peer_agent</i> state during get operation
<i>known_nodes</i>	Simulates external knowledge about existing nodes
<i>key_value</i>	Select a $\langle key, value \rangle$ for storing in the network
<i>keys</i>	Select to look for a <i>value</i> with particular <i>key</i>

TABLE 1. Chord functions

The *hash* function assigns identifiers of nodes to peers and keys:

- $hash : Peer \cup Key \rightarrow Chord \cup \{undef\}$ ,

where *undef* is a special value which indicates that:

- there are  $N$  nodes in the network, and an identifier is requested for a new node, or
- there are  $N$  keys in the network, but we try to add a new key.

The function must also guarantee that in each moment two different active peers (keys) have different hash values. However, note that it is possible that in different moments different peers have the same identifier. Also, it may happen that a peer can have different identifiers (obtained by different calls of the *hash* function before and after a period in which the peer is not present in the network). The above mentioned *id* function can be explained as a "local" counterpart of *hash*. Namely, we can assume that the values produced by *hash* are stored in the local memory and read and published by *id* to reduce the number of expensive calls of *hash*. In the program given below, *id* will be invoked with the argument *Me* to allow a node to identify itself in the network.

The external function *ping*, defined as:

- $ping : Chord \rightarrow \{true, false\}$

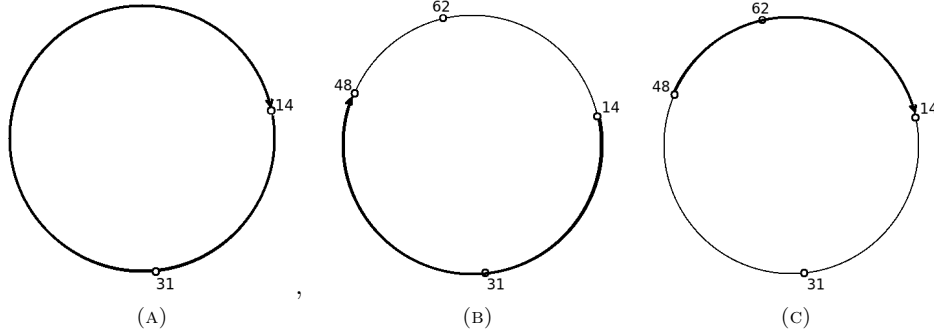
returns *true* or *false*, depending on whether the argument is reachable in the network.

The function *member\_of*:

- $member\_of : Chord \times Chord \times Chord \rightarrow \{true, false\}$ ,

determines whether the first argument is between two next two arguments with respect to the ring ordering, more formally:

- if  $arg_2 = arg_3$  always returns *true*,
- if  $arg_2 < arg_3$  returns *true* if  $arg_2 < arg_1 \leq arg_3$  holds,
- if  $arg_2 > arg_3$  returns *true* if  $\neg(arg_3 \leq arg_1 < arg_2)$  holds,
- otherwise returns *false*.

FIGURE 2. *member\_of*

**Example 4.1.** Let  $N = 64$ . Then,

- $member\_of(31, 14, 14) = true$ , see Fig. 2.a,
- $member\_of(31, 14, 48) = true$ , see Fig. 2.b,
- $member\_of(62, 14, 48) = false$ , see Fig. 2.b,
- $member\_of(31, 48, 14) = false$ , see Fig. 2.c,
- $member\_of(62, 48, 14) = true$ , see Fig. 2.c.

□

In this paper we will realize communication following the ideas from [3] and [5]. During communication a direct channel is open between two nodes (assuming that channels do not lose information):

- $communication : Chord \times Chord \rightarrow Message^*$ .

The first argument is the *sender* and the second one is the *receiver* of a message. When a *sender* sends a new message to a *receiver*, it appends it to the list of all messages which were sent by this *sender* to that *receiver*. When the *receiver* processes this message, it removes it from the list. Different type of messages contain different information:

- if the type of communication is *request\_and\_remove\_keyvalue*, the message is empty,
- if the type of communication is *send\_keys*, the message contains all *keyvalue*-pairs that belong to the sender
- if the type of communication is *set\_predecessor*, the message is empty
- if the type of communication is *get\_predecessor*, the message contains *predecessor* of the sender
- if the type of communication is *find\_successor*, the message contains two values - the value for which the successor is asked, and the *id* of the peer which initiated the query
- if the type of communication is *get*, the message contains the hashed value of the key
- if the type of communication is *successor\_found*, the message contains the value for which successor was asked and information about the successor
- if the type of communication is *value\_found*, *received\_keyvalue* or *received\_predecessor*, the message contains the resulting information.

The function *mode*:



- $mode : Peer \rightarrow Mode$

determines *Peer\_agent* state. Initially, for all  $p \in Peer$  value of  $mode(p)$  is set to *not\_connected*.

The functions *mode\_join*, *mode\_leave*, *mode\_stabilize*, *mode\_fingers*, *mode\_put* and *mode\_get*:

- $mode\_join : Peer \rightarrow Mode\_join$
- $mode\_leave : Peer \rightarrow Mode\_leave$
- $mode\_stabilize : Peer \rightarrow Mode\_stabilize$
- $mode\_fingers : Peer \rightarrow Mode\_fingers$
- $mode\_put : Peer \rightarrow Mode\_put$
- $mode\_get : Peer \rightarrow Mode\_get$

determine *Peer\_agent* states during some of the Chord operations. Initially, for all  $p \in Peer$  the values of these functions are set to *undef*.

The external function *known\_nodes*:

- $known\_nodes : Peer \rightarrow Chord$

simulates external knowledge about the nodes in the particular Chord network.

The external function *key\_value*:

- $key\_value : Peer \rightarrow Key \times Value$

simulates the choice of a node to store a  $\langle key, value \rangle$  pair in the Chord network.

The external function *keys*:

- $keys : Peer \rightarrow Key$

simulates the choice of a node to look if some *value* with particular *key* is stored in the Chord network.

**4.2. Chord Rules.** The rest of this section contains our ASM-formalization of the Chord protocol. We present the general program executed by every peer, and a high level description of the rules performed in a Chord network which corresponds to the pseudo code given in [22] (note that the rules FAIRLEAVE, UNFAIRLEAVE, PUT and GET are not given there). A detailed specification of these rules is provided in Appendix A.

**4.2.1. Peer\_agent Module.** The following main module contains actions that are executed by every peer. The mode of all peers is initially *not\_connected*. After a node joins a network successfully, its mode is changed to *connected*. In each execution of a loop, a node concurrently calls the rules responsible for the ring topology maintenance (STABILIZE, UPDATEPREDECESSOR, UPDATEFINGERS) and communication (READMESSAGES) and, according to a non-deterministic choice, it might also invoke one of the FAIRLEAVE, UNFAIRLEAVE, PUT and GET rules.

```

if  $mode(Me) = not\_connected$  then
  if Choosed Action Is Join
    seq
      if There Are No Known Nodes then
        START
      else
        JOIN
      endif
    if Connection Successful then

```

```

    mode(Me) := connected
  else
    mode(Me) := not_connected
  endif
endseq
endif
else
  if mode(Me) = connected then
    if id(Me) Does Not Have Communication Problems then
      par
        READMESSAGES
        STABILIZE
        UPDATEPREDECESSOR
        UPDATEFINGERS
        ExtendedJoinModel =
          seq
            choose action in Action
          par
            LeavingActions =
              FAIRLEAVE Or UNFAIRLEAVE
            KeyValueHandling =
              PUT Or GET
          endpar
        endseq
      endpar
    else
      mode(Me) := not_connected
    endif
  endif
endif
endif

```

4.2.2. *Chord Rules - High Level Description.* In the sequel, we will describe the rules of the Chord protocol (listed in Table 2).

When  $Node_i$  starts a new *Chord network*, the following rule is executed:

```

START=
seq
  id(Me) := hash(Me)
  Initialize Values For Node id(Me)
endseq

```

When  $Node_i$  asks  $Node_j$ , which is the member of a *Chord network*, to join the network the following rule is executed (a network is fulfilled if it already contains  $N$  nodes):

```

JOIN=
seq
  id(Me) := hash(Me)

```

Rule	Description	Resulting State
START	The first node starts the network	A state with one node
JOIN	A new node joins the network	A state with an additional node
FAIRLEAVE	A node leaves fairly the network	A state without one node
UNFAIRLEAVE	A node leaves/crashes	A state without one node
STABILIZE	A node updates its successor and predecessor	Successor and predecessor update
UPDATEPREDECESSOR	Periodic check of the predecessor	Predecessor update
UPDATEFINGERS	A node runs update on its finger table	Updating finger table entries
PUT	A new (key, value) pair is stored	Updating (key,value) table
GET	Finding a value for a given key	Unchanged state
FINDSUCCESSOR	Finding a responsible node for given key or successor of a node	Unchanged state
READMESSAGE	Read messages dedicated to a node	Changing some local variables if it is requested

TABLE 2. Chord rules

```

if Chord Is Not Fulfilled then
  seq
    Initialize Values For Node  $id(Me)$  With Respect To  $known$ 
    Take  $keyvalue$  That Should Belong To
      Node  $id(Me)$  From  $successor(id(Me))$ 
    endseq
  endif
endseq

```

After an application of the join rule, all keys which have the hash value less or equal then the  $id$  of the new node, and which have been stored in the  $keyvalue$  list of the successor of the new node are moved to the  $keyvalue$  list of the new node.

When  $Node_i$  leaves a *Chord network* in a fair way, it executes the following rule:

```

FAIRLEAVE=
seq
  if  $successor(id(Me))$  Is Not Alive then Update  $successor(id(Me))$ 
  endif
  if  $id(Me)$  Is Not The Only Node In Chord then
    seq
      par
        Give  $keyvalue(id(Me))$  To  $successor(id(Me))$ 
        Remove  $id(Me)$  From Successor Pointers Ring
      endpar
      Deactivate Values For Node  $id(Me)$ 
    endseq
  endif
endseq

```

Note that a node can also leave a *Chord network* in an unfair way, for example caused by a node crash, communication problems, etc. As it will be shown in Section 5 the other rules can detect such situations. Thus, the body of UNFAIRLEAVE rule is empty.

The next three rules are responsible for detecting situations in which the considered *Chord network* is not in a stable state, updating information about the network and reconnecting the successor pointers of nodes to establish a stable state. These rules are applied periodically, while a peer belongs to the considered *Chord network*:

STABILIZE=

```

if successor(id(Me)) Is Alive then
  seq
    Set x To Be predecessor(successor(id(Me)))
    if (x ≠ undef ∧ member_of(x, id(Me), successor(id(Me)))) then
      seq
        successor(id(Me)) := x
        Take Keyvalue That Should Belong To
        Node id(Me) From successor(id(Me))
      endseq
    endif
    if x = undef ∨ (x ≠ undef ∧ member_of(id(Me), x, successor(id(Me)))) then
      Notify successor(id(Me)) To Update Its predecessor To id(Me)
    endif
  endseq
else
  Update successor(id(Me))
endif

```

STABILIZE is responsible to update information on successor of a node either because the successor is crashed or the new node with appropriate *id* has enter the network, and to notify new successor about its new predecessor.

Checking if the current predecessor of a node is still active is realized by:

UPDATEPREDECESSOR=

```

if predecessor(id(Me)) Is Not Alive then
  Deactivate Values For predecessor(id(Me))
endif

```

Updating the values from finger table is realized by firing:

UPDATEFINGERS=

```

For Next next(id(Me)) Update finger.listitem(next(id(Me)))
  With Responsible Node For id(Me) ⊕N 2next(id(Me))-1

```

During one cycle of *Peer\_agent Module* exactly one value from finger table is updated.

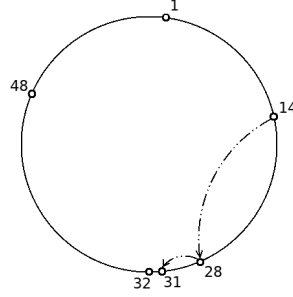
Storing a new  $\langle key, value \rangle$  pair is realized by the following rule:

PUT=

```

if Chord Is Not Fulfilled then
  Notify Responsible Node For hash(key) To Add ⟨hash(key), value⟩

```

FIGURE 3. *find\_successor*

```

To Its keyvalue Table
endif

```

The name of the rule `FINDSUCCESSOR` comes from [20]. By this rule a node is asked to return successor of the given argument. The corresponding argument can be the hash value of a key, or the *id* of a peer. In the former case, the result is the identifier of the member of the network which is responsible for the  $\langle key, value \rangle$  pair. In the later case, the rule gives the identifier of the first member of the network which is equal to, or greater then, the argument. Note that, if the argument is the *id* of a node (i.e., a peer that is active in a network), the result is *id*, and not the identifier of its successor.

```

FINDSUCCESSOR=
For Given key
  if member_of(key, id(Me), successor(id(Me))) then
    Respond With successor(id(Me))
  else
    Forward Query To Closes Predecessor From finger(id(Me))
  endif

```

In the above rule, for the argument  $h$ , the current node  $n$  returns its successor to the node which was started the query, if  $member\_of(h, n, successor(n)) = true$ . Otherwise, the query is forwarded to the node whose *id* precedes  $h$  in  $n$ 's finger table, or is the maximal element of this table.

**Example 4.2.** Let  $N = 64$  and a Chord network contain the nodes  $\{1, 14, 28, 31, 32, 48\}$  (see Fig. 3). Let the corresponding successor pointers form a ring. Note that there is no node with the identifier 29 in the network. The result of call of the successor of 29 by the node 14 is 31, because node 29 is understood as the hash of a key or the *id* of a node entering the network. The node 14 will transfer the query to its successor (node 28). This node will return its successor (node 31) as the result, because  $member\_of(29, 28, 31)$  is *true*.

However, since the node with the identifier 31 exists in the network, the result of the call of the successor of 31 will be 31.

Finally, note that, if one needs information on the successor of a node, for example the node 31, then it should call for a successor of  $31 \oplus_{64} 1$ .  $\square$

Any node present in a Chord network can execute GET rule (ask for the value of a key). That rule does not change the actual state of the network, but we define it as:

GET=  
 Invoke FINDSUCCESSOR For Given *key*  
 And Check Corresponding *value*

During the each execution of a *Peer\_agent Module* all the messages send to a node are processed:

READMESSAGES=  
 Read Messages Dedicated To *Me*,  
 Change Local Variables If It Is Requested And  
 Clear Processed Messages

## 5. CORRECTNESS OF THE FORMALIZATION

In this section we present the correctness of our formalization with respect to the so-called regular runs. First, following [11] we formally consider a distributive algebra  $\mathcal{A}$  which consists of the module *Peer\_agent* introduced in Section 4.2, a vocabulary  $\Upsilon$ , and a collection of  $\Upsilon$ -states, such that:

- the *vocabulary*  $\Upsilon$  contains:
  - all function names that appear in the module *Peer\_agent*, except *Me*,
  - a nullary function name *undef*, the standard Boolean constants and operations, and
  - a unary function name *Mod*,
- a *state*  $S$  of the *vocabulary*  $\Upsilon$  is a pair which consists of:
  - a base set  $Peer \cup Key \cup Value \cup Chord \cup Action \cup Communication \cup \{Peer\_agent\}$ , where *Peer\_agent* is a module name, and
  - an  $\Upsilon$ -interpretation  $I$  which satisfies that  $I(Mod) : Peer \rightarrow \{Peer\_agent\}$ , while the other function names are interpreted as the corresponding objects defined at the beginning of Section 4,
- in a local state (of the *vocabulary*  $\Upsilon \cup \{Me\}$ ) which corresponds to the peer  $p \in Peer$  and the state  $S$ , denoted by  $View(p; S)$ , all symbols are interpreted as in  $S$ , and additionally  $I(Me) = p$ .

In the initial state a value  $mode(p)$  is set to *not\_connected* and  $id(p)$  is set to *undef* for all  $p \in Peer$ , while values of  $successor(c)$ ,  $predecessor(c)$ ,  $finger(c)$ ,  $next(c)$  and  $keyvalue(c)$  are set to *undef* for all  $c \in Chord$ .

**Definition 5.1.** Let  $x_1, x_2 \in Node$  and  $y_0, \dots, y_r \in Node$  be all the nodes from a Chord network such that  $x_1 = y_0 < \dots < y_r = x_2$ . The pair  $\langle x_1, x_2 \rangle$  forms a *stable pair* in a state if the following holds:

- $y_{i+1} = successor(y_i)$ ,  $y_i = predecessor(y_{i+1})$ , for all  $i \in \{0, \dots, r-1\}$ .

A Chord network  $\{x_0, \dots, x_{k-1}\}$ ,  $k \geq 1$ , is *stable* in a state if the pair  $\langle x_0, x_0 \rangle$  is stable.  $\square$

Intuitively, a pair  $\langle x_1, x_2 \rangle$  is stable in a state if there is no node trying to join the network through the node on the ring-interval  $(x_1, x_2)$  in that state.

A move at a state  $S$  is an execution of the module *Peer<sub>agent</sub>* by a peer  $p$  at  $View(p; S)$ . The corresponding rules belong to the low-level description given in Appendix A. Similarly as in [11], we consider only atomic moves.

A run<sup>1</sup> of our distributive algebra  $\mathcal{A}$  is a triple  $\langle Moves, P, \sigma \rangle$  such that:

- $Moves$  is a partially ordered set of moves of peers such that every set  $\{y : y \leq x\}$  is finite, and  $y < x$  means that the move  $y$  must be finished before  $x$  begins,
- the function  $P$  associates with every  $x \in Moves$  a peer  $P(x)$  which executes  $x$ , and satisfies that each nonempty set  $\{x : P(x) = p\}$  is linearly ordered, and
- the function  $\sigma$  associates with every finite initial segment of  $Moves$  a state of  $\mathcal{A}$ , while  $\sigma(\emptyset)$  denotes an initial state.

Note that the assumption that every set  $\{y : y \leq x\}$  is finite implies the fairness, in the sense that every node will eventually execute its next move. Furthermore, we require that the following coherence condition holds for every run:

- Let  $x$  be a maximal element of a finite initial segment  $X$  of  $Moves$ , and  $Y = X \setminus \{x\}$ . Then,  $x$  transforms  $\sigma(Y)$  to  $\sigma(X)$ .

A state is reachable in a run  $\langle Moves, P, \sigma \rangle$  if it belongs to the range of  $\sigma$ .

A run  $\rho' = \langle Moves', P', \sigma' \rangle$  is an initial segment of a run  $\rho = \langle Moves, P, \sigma \rangle$  if  $Moves'$  is an initial segment of  $Moves$  and  $P'$  and  $\sigma'$  are restrictions of  $P$  and  $\sigma$ , respectively. A run  $\rho'$  is a linearization of a run  $\rho$  if  $Moves'$  is a linearization<sup>2</sup> of  $Moves$ ,  $P'$  and  $P$  coincide, and  $\sigma'$  is a restriction of  $\sigma$ . Note that each linearization is a sequential run. Two fundamental corollaries formulated in [11] are:

- Corollary 5.1.** (1) All linearizations of the same finite initial segment of a run have the same final state.
- (2) A property holds in every reachable state of a run  $\rho$  iff it holds in every reachable state of every linearization of  $\rho$ .

**Definition 5.2.** *Regular runs* are all runs of a distributive algebra  $\mathcal{A}$  which satisfy that:

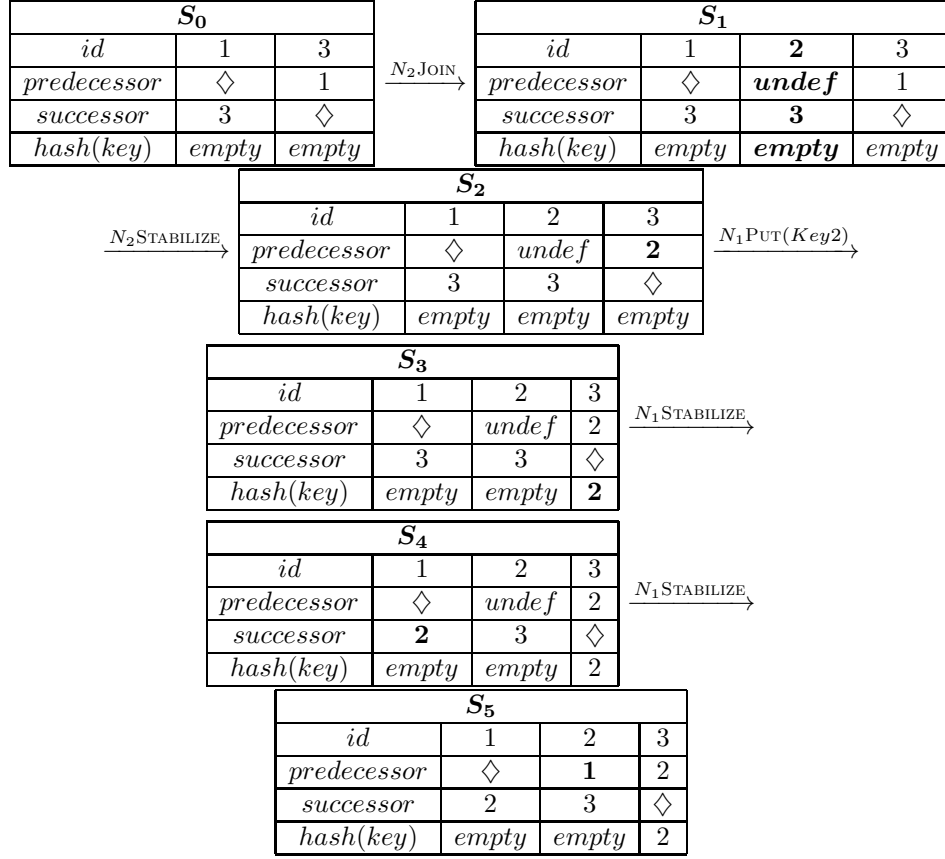
- any execution of FAIRLEAVE, UNFAIRLEAVE and PUT might happen only between a stable pair of nodes.  $\square$

The following example illustrates the need for the above constraint. In the example and in the rest of the paper we will graphically illustrate sequences of moves, so that  $S_i$  denotes a state, the updated values are in bold, and  $\diamond$  means that the rest of a network is not affected by a move.

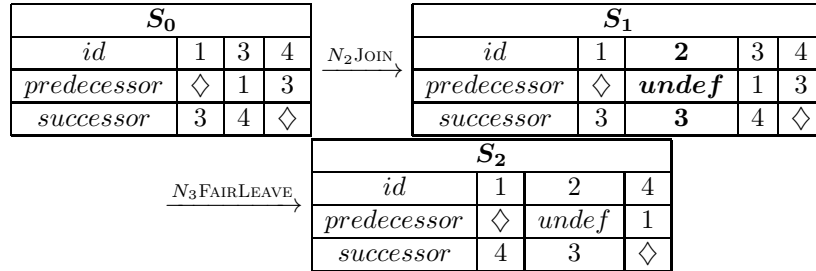
**Example 5.1.** Let  $S_0$  be the initial state in which the nodes  $N_1$  and  $N_3$  are members of a network, and the node  $N_2$  wants to join. Suppose that before the pair  $\langle N_1, N_3 \rangle$  becomes stable,  $N_1$  executes the put rule with the hash 2 of a key. Since  $N_1$  is not aware of  $N_2$ , the corresponding key will be stored in  $N_3$ , and not in  $N_2$ .

<sup>1</sup>Note that this is a simplification of the corresponding definition from [11] which results from the fact that the set *Peer* is fixed at the beginning and cannot be changed during executions of Chord.

<sup>2</sup> $Moves'$  is linearly ordered,  $Moves'$  and  $Moves$  have the same elements, and if  $x < y$  in  $Moves$ , then  $x < y$  in  $Moves'$ .



Again, assume that  $S_0$  is the initial state and the network contains the nodes  $N_1$ ,  $N_3$  and  $N_4$ . If the node  $N_2$  executes the join rule, and before the pair  $\langle N_1, N_4 \rangle$  becomes stable,  $N_3$  wants to leave,  $N_2$  will be isolated from the rest of the network, and the other nodes will never be aware of it.



A similar example can be given for UNFAIRLEAVE. □

In the sequel, we show that a stable pair of nodes in a Chord network, which executes a regular run, eventually becomes stable after adding/removing of a node between them (the theorems 5.1-5.6). Corollary 5.3 formulates the corresponding statement for a stable network. Finally, we prove that the proposed key-handling correctly distributes keys and answers queries (Theorem 5.7 and Corollary 5.4).

The first theorem expresses that the rule FINDSUCCESSOR will terminate in a finite number of steps. It corresponds to Theorem IV.2 from [20–22].



**Theorem 5.1.** Let  $n \in Chord$  be the node which fires the rule FINDSUCCESSOR for  $h \in \{0, 1, \dots, N - 1\}$ . Let  $m'$  be the minimal element of  $Chord$  such that  $h \leq m'$ . If the pair  $\langle n, m' \rangle$  is stable in that state, the node  $n$  will get the result after a finite number of moves.

*Proof.* Let the node  $n$  be asked for the successor of  $h$ :

- if  $h$  is a member of the ring-interval  $(n, successor(n)]$ ,  $successor(n)$  as the result is returned; otherwise
- a node  $k$  (such that the ring-interval  $(k, h]$  is the smallest subset of the ring-interval  $(n, h]$  for every node from the finger table of  $n$ ) is chosen to answer the query.

Then, FINDSUCCESSOR is invoked by the node  $k$ . The number of such calls is limited since there are finitely many nodes between  $n$  and  $h$ . Furthermore, in each step the ring-interval  $(k, h]$  shrinks, and since - by the construction - the first element of the finger table of a node is its successor, we will eventually reach the node  $m$  such that  $h \in (m, successor(m)]$ . Note that in each step a node (an active member of the network) is contacted, and that the result is directly sent to the node  $n$  which issued the query. Node  $n$  must be active at that moment. It cannot execute (UN)FAIRLEAVE before it gets the answer, since (UN)FAIRLEAVE actions can be executed only when nodes do not wait for answers from earlier fired queries.  $\square$

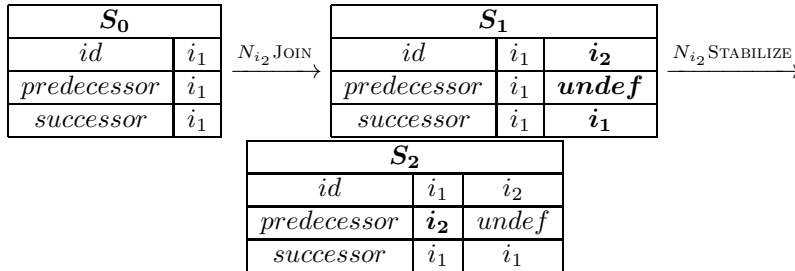
Theorems 5.2 – 5.6 guarantee that the successor and predecessor pointers for each node will be eventually up to date after a node joins, or unfair leaves the network. In the corresponding proofs we will use some finite initial sequences of runs. Due to the fact that the STABILIZE and UPDATEPREDECESSOR are applied periodically by all nodes in a network, we will mention only those applications which change the values of the functions *predecessor* and *successor*.

Note that, in each proof we will consider some fixed linearization of moves, but according to Corollary 5.1, all linearizations of the corresponding regular run will result in the same final state.

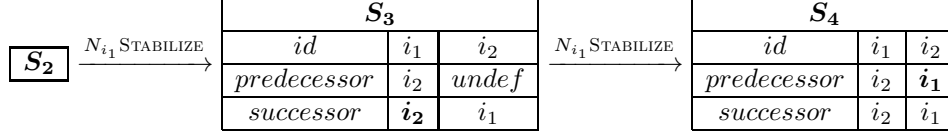
Theorem 5.3 corresponds to Theorem IV.3 from [20–22].

**Theorem 5.2.** Let a peer join a Chord network, between two nodes which constitute a stable pair. Then, there is a number  $k > 0$  of steps, such that if no other join rule happens in the meantime, the STABILIZE rule will bring the starting pair to be stable after  $k$  steps.

*Proof.* Suppose that the network contains only one node  $N_{i_1}$  and that  $N_{i_2}$  wants to join. We will consider the following sequence of moves:

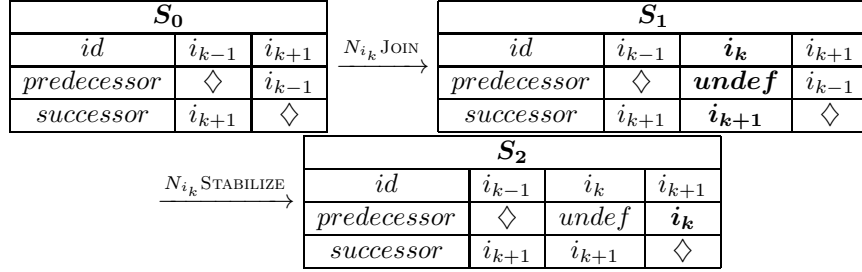


At this moment the pair is not stable, and the restriction to the regular runs does not allow the nodes to try to leave the network. Also, according to the assumption of the statement, other peers will not execute the JOIN rule. Then, we have:

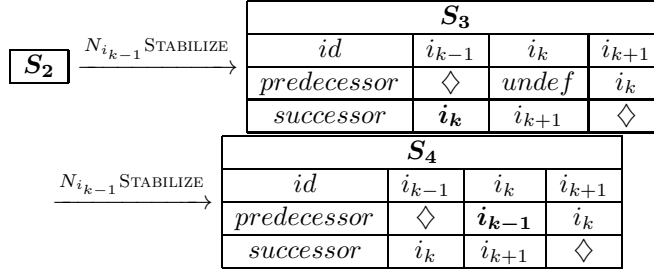


Obviously, a stable pair has been established, again.

Suppose that there are two or more nodes in the network, and that  $N_{i_k}$  wants to join. Let  $N_{i_{k-1}}$  and  $N_{i_{k+1}}$  be the members of the network such that  $\text{successor}(i_{k-1}) = i_{k+1}$ , and  $\text{predecessor}(i_{k+1}) = i_{k-1}$  (i.e.,  $\langle N_{i_{k-1}}, N_{i_{k+1}} \rangle$  is a stable pair). Furthermore, let  $\text{member\_of}(i_k, i_{k-1}, i_{k+1}) = \text{true}$ . Then, we consider the following sequence of moves:



Similarly as above, the restriction to the regular runs does not allow the nodes to try to leave the network, other peers will not execute the JOIN rule, and we have:

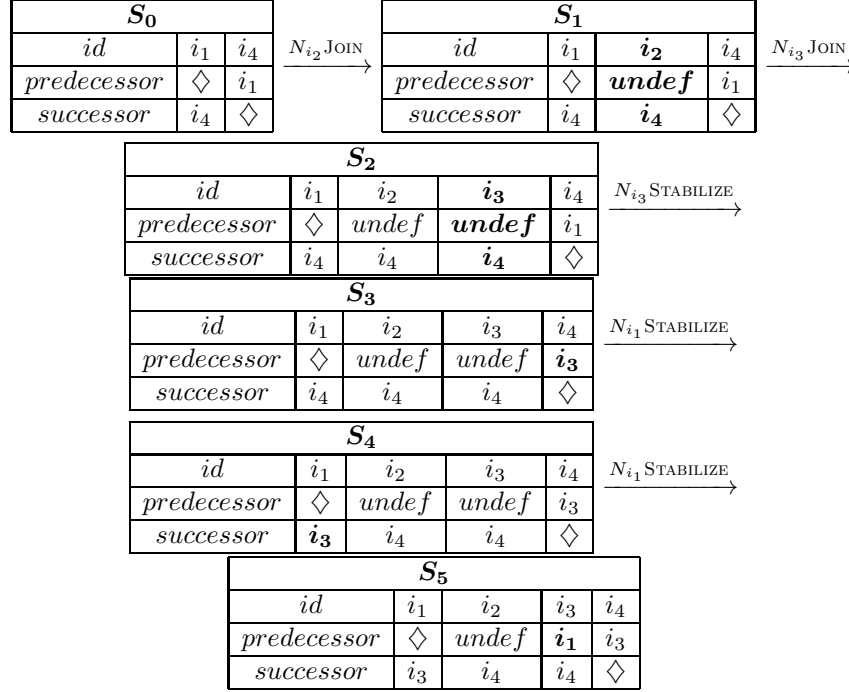


Thus, a stable pair has been established. □

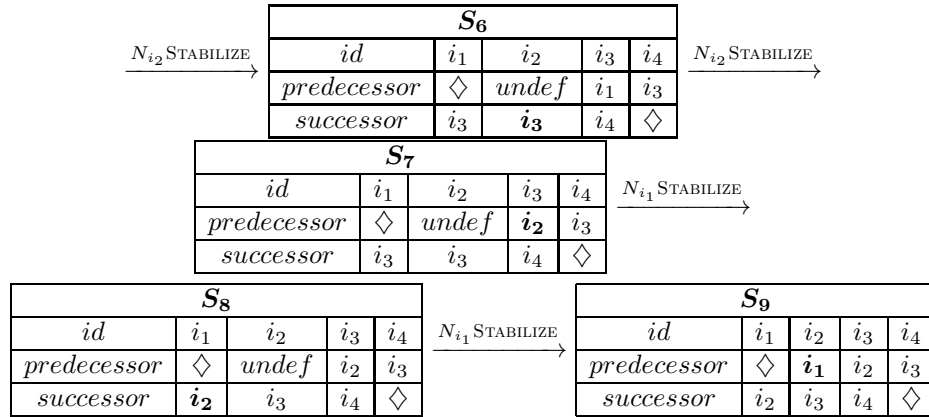
**Theorem 5.3** (Concurrent joins). Let a Chord network contain a stable pair. If a sequence of JOIN rules is executed between the nodes which form this stable pair, interleaved with STABILIZE, UPDATEPREDECESSOR and UPDATEFINGERS, then there is a number  $k > 0$  of steps, such that after the last JOIN rule, the starting pair of nodes will be stable after  $k$  steps.

*Proof.* First, note that UPDATEFINGERS does not change the values of the functions *predecessor* and *successor*. Similarly, UPDATEPREDECESSOR might change values of the function *predecessor* only after an UNFAIRLEAVE. Thus, we do not consider executions of UPDATEPREDECESSOR and UPDATEFINGERS in the rest of this proof.

Let us assume that all peers that want to join the network have different successors. Then, by Theorem 5.2, the statement holds. Otherwise, there must be at least two peers that want to join the network having the same successor. Suppose that  $N_{i_2}$  and  $N_{i_3}$  want to join and that  $N_{i_1}$  and  $N_{i_4}$  are members of the network, such that  $successor(i_1) = i_4$  and  $predecessor(i_4) = i_1$ . Furthermore, let  $member\_of(i_2, i_1, i_3) = true$  and  $member\_of(i_3, i_2, i_4) = true$ . Then, we consider the following sequence of moves:



At this moment, the successor pointers of  $N_{i_1}$  and  $N_{i_3}$  connect those nodes and  $N_{i_4}$ . Then, similarly as in Theorem 5.2, executions of STABILIZE by  $N_{i_2}$  and  $N_{i_1}$  result with the stable pair  $\langle N_{i_1}, N_{i_4} \rangle$  in the state  $S_9$ :



The execution of the STABILIZE rule by  $N_{i_3}$  in  $S_2$  does not change the values  $predecessor(i_2)$  and  $successor(i_2)$ . The same holds if more than one node (denoted

$N_{j_1}, N_{j_2}, \dots$ ) join the network between  $N_{i_1}$  and  $N_{i_3}$ . So, let us assume that

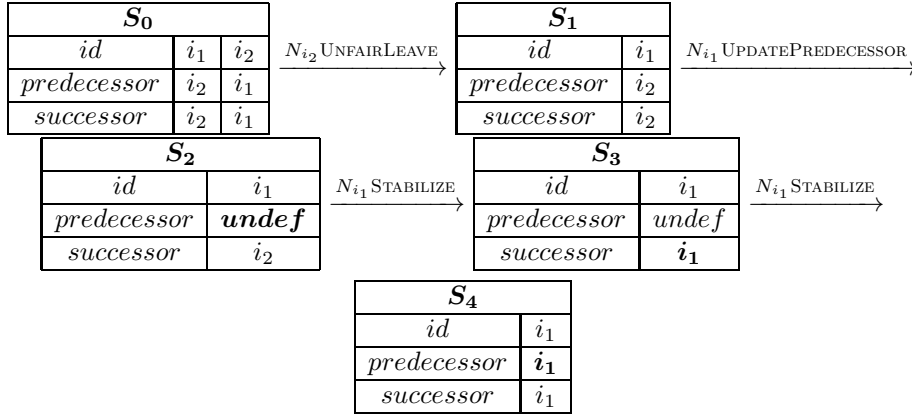
$$i_1 \leq \dots \leq j_2 \leq j_1 \leq i_3 \leq i_4.$$

Then, similarly as above, a sequence of executions of the STABILIZE rule by the nodes  $N_{j_1}, N_{i_1}, N_{j_2}, N_{i_1}, \dots$  result with a stable starting pair.  $\square$

**Theorem 5.4.** Let a Chord network contain a stable pair and let a node between them leave the network. Then, there is a number  $k \geq 0$  of steps, such that if no JOIN rule happens at the considered part of the network in the meantime, the pair will be brought into a stable state after  $k$  steps.

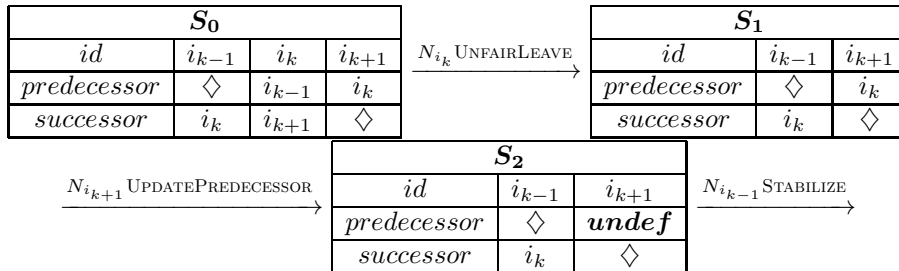
*Proof.* Let us first assume that the node leaves the network in a fair way. Since FAIRLEAVE produces a stable pair, the statement holds for  $k = 0$ . Thus, let UNFAIRLEAVE be executed.

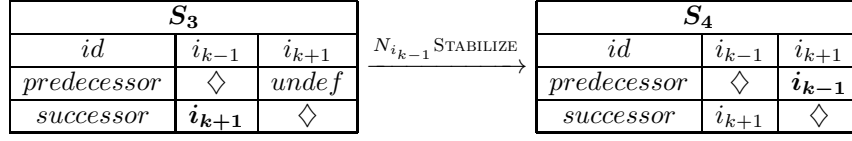
Suppose that the network contains only two nodes  $N_{i_1}$  and  $N_{i_2}$ , and that  $N_{i_2}$  leaves in an unfair way and breaks the ring. Then, we consider the following sequence of moves:



The state  $S_4$  is stable.

Suppose that there are three or more nodes in a network. Let  $N_{i_{k-1}}, N_{i_k}$  and  $N_{i_{k+1}}$  be the members of the network such that  $successor(i_{k-1}) = i_k$  and  $successor(i_k) = i_{k+1}$ . Suppose that  $N_{i_k}$  unfair leaves and breaks the ring of the successors pointers. Then, we consider the following sequence of moves which results with the stable pair  $\langle N_{i_{k-1}}, N_{i_{k+1}} \rangle$  in state  $S_4$ :

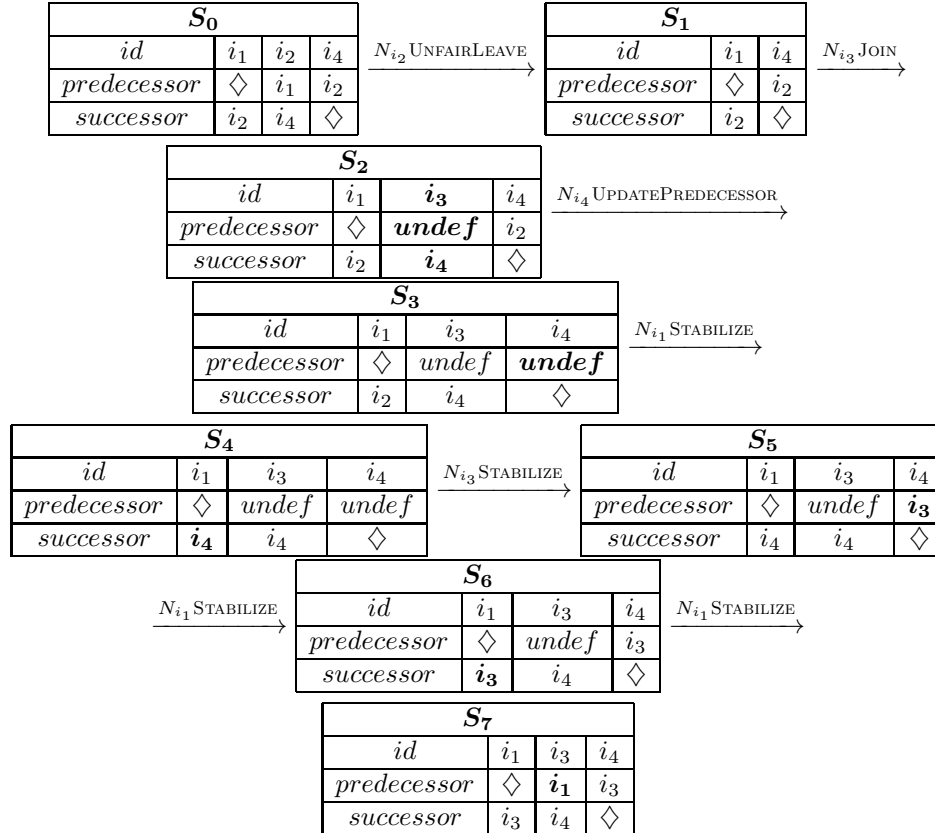




□

**Theorem 5.5.** Let a Chord network contain a stable pair. Let a node which is between those nodes leave the network following by several nodes which want to join between them. Then, there is a number  $k \geq 0$  of steps, such that the considered pair will be brought into a stable state after  $k$  steps.

*Proof.* Let us first assume that the node leaves the network in a fair way. It produces a stable pair, and according to the theorems 5.3 and 5.4, the statement holds. Then, let the node  $N_{i_2}$  execute UNFAIRLEAVE and break the ring. If no node joins the network in the ring interval  $[predecessor(i_2), successor(i_2)]$ , the statement holds similarly as in theorems 5.3 and 5.4. Finally, assume that a node joins the network in the ring interval  $[predecessor(i_2), successor(i_2)]$ . Suppose that  $N_{i_1}$ ,  $N_{i_2}$  and  $N_{i_4}$  are members of the network, such that  $successor(i_1) = i_2$ ,  $predecessor(i_2) = i_1$ ,  $successor(i_2) = i_4$  and  $predecessor(i_4) = i_2$ . Furthermore, let  $member\_of(i_2, i_1, i_3) = true$  and  $member\_of(i_3, i_2, i_4) = true$ . Let  $N_{i_2}$  be the node that will leave, and  $N_{i_3}$  node that will join the network. We consider the following sequence of moves:



which results with the stable starting pair in the state  $S_7$ . If more than one node want to join the network in the considered ring interval, we can use similar arguments as in the proof of Theorem 5.3 to establish the statement.  $\square$

Note that the restriction from the formulation of Theorem 5.5, that no other leave-rules are allowed after the first one, is not essential. According to the definition of regular runs, leave-rules can be executed only between nodes which constitute a stable pair, and we can consider an execution of a sequence of join rules interleaved with leave-rules, and obtain the same result. The above statement will hold for each subsequence which starts with a leave rule followed by several join rules. Thus, we have the following:

**Corollary 5.2.** Let a Chord network contain a stable pair. Let a node, which is in between those nodes, leave the network. Then, there is a number  $k \geq 0$ , such that the considered pair of nodes will become stable after  $k$  moves.

Theorem 5.6 incorporates all previous ideas, and is the main statement concerning correctness of maintaining topological structure of Chord networks.

**Theorem 5.6.** Let a finite initial segment of a run produce the state  $S$  of a Chord network. Then, for every pair of nodes  $n, n' \in \text{Chord}$ , there is a number  $k \geq 0$ , such that  $\langle n, n' \rangle$  will become stable after  $k$  moves.

*Proof.* The case in which the state  $S$  is stable is trivial. So, let us assume that  $S$  is not stable. According to the definition of regular runs, no leave rule might happen before the network becomes stable. Thus, only a sequence of JOIN rules interleaved with STABILIZE, UPDATEPREDECESSOR and UPDATEFINGERS can be executed. Since the number of nodes in the network is limited, the number of join rules in the sequence must be finite. Then, similarly as in Theorem 5.5, the statement holds.  $\square$

Since a network is stable in a state if all pairs of nodes from the network are stable in that state, we have:

**Corollary 5.3.** Let a finite initial segment of a run produce the state  $S$  of a Chord network. Then, there is a number  $k \geq 0$ , such that the network will become stable after  $k$  moves.

Finally, the next two statements say that our formalization consistently manipulates distributed keys. Theorem 5.7 states that  $(key, value)$  pairs are properly distributed over the network. Informally, it follows from the facts that for every  $n \in \text{Chord}$ ,  $hash(key) \leq n$  for the keys for which  $n$  is responsible for, and that all rules that manipulate  $(key, value)$  pairs invoke FINDSUCCESSOR rule.

**Theorem 5.7** (Golden rule).

$$\begin{aligned} \forall((key, value) \in Keys \times Values, n \in \text{Chord})((key, value) \in keyvalue(n) \\ \Rightarrow member\_of(hash(key), predecessor(n), n)). \end{aligned}$$

*Proof.* Obviously, the statement holds for a Chord network containing only one node. According to the definition, a JOIN executed by  $N_i$  moves to  $N_i$  the corresponding  $(key, value)$  pairs stored in its successor. Similarly, FAIRLEAVE and STABILIZE executed by  $N_i$  transfer all  $(key, value)$  pairs from  $N_i$  to its successor. When a node leaves the network in a unfair way, all  $(key, value)$  pairs it stored

would be lost. Thus the statement remains true. By the definition, for a given  $(key, value)$  pair, PUT finds a node responsible for the pair which satisfies the statement. Finally, all the other rules do not manipulate  $(key, value)$  pairs.  $\square$

Corollary 5.4 follows from the definition of GET, and the theorems 5.1 and 5.7:

**Corollary 5.4.** If GET returns *undef* for some  $key \in Keys$ , then there is no  $value \in Values$  such that  $(key, value)$  pair is stored in the Chord network.

Namely, according to Theorem 5.7, all  $(key, value)$  pairs are stored properly, and from Theorem 5.1 GET considers only the  $(key, value)$  pairs stored in the node  $N$  which satisfy condition  $member\_of(hash(key), predecessor(id(N)), id(N))$ .

## 6. DISCUSSION AND RELATED WORKS

One of the basic ideas behind the results presented in Section 5 is that we consider not all possible, but only regular runs. Example 5.1 shows that without that restriction the ring topology can be damaged which might result in splitting a Chord network in disjoint parts and/or in losing data.

The request that every execution of the module *Peer\_agent* by a peer is atomic can be relaxed so that only executions of the Chord-rules are required to be atomic. In that case it is necessary to introduce, so called, Time-To-Leave (TTL) mechanism. It means that if some rule does not finish in predefined number of module executions, a node repeats execution of that rule from the beginning.

On the other hand, the atomicity assumption is essential and cannot be completely avoided since it eliminates several observed shortages, for instance when a node tries to join the existing network via another node which tries to leave in the same moment. Another counterexample concerns a Chord-network and a node  $N_i$  which leaves the network by executing the FAIRLEAVE rule. Then,  $N_i$  tries to transfer its *keyvalue*-table to  $N_{successor(i)}$ . However, suppose that  $N_{successor(i)}$  leaves the network in the same moment. Obviously, the content of  $N_i$ 's *keyvalue*-table will be lost.

Several improvements of Chord protocol are proposed to increase robustness:

- in [20] each Chord node can maintain a list containing the node's first  $r$  successors, and
- it is usual in software implementations [10] to make multiple copies of all  $(key, value)$ -pairs (for example, by different hash functions).

Note that those changes might improve performance of a Chord network, but do not affect correctness. Quite simple generalizations of Example 5.1 would produce incorrect behavior.

The Chord protocol is introduced in [20–22]. The papers analyze the protocol, its performance and robustness, under the assumption that the nodes and keys are randomly chosen and give several theorems that involve the phrase *with high probability*, for example: "With high probability, the number of nodes that must be contacted to find a successor in a  $N$ -node network is  $O(\log N)$ " [22, Theorem IV.2]. Our Theorem 5.1 corresponds to that statement, but only proves correctness of the rule FINDSUCCESSOR. Theorem [22, Theorem IV.2] considers the so-called pure join model of Chord which does not allow (un)fair-leaving, while we permit failures of nodes, but restrict runs to be regular. If we assume the random distribution, it is easy to see that the same complexity result can be obtained. On the other

hand, in the theorems [22, Theorem IV.5] and [22, Theorem IV.6] behavior of FINDSUCCESSOR is analyzed when failures of nodes can happen. Then, to prove high probability of correctness it is necessary to involve the aforementioned improvement of Chord which concerns lists of successors of nodes. The only statement in [22] which avoids the mentioned phrase about high probability is Theorem IV.3. It (corresponds to our Theorem 5.3 and) proves that inconsistent states produced by executing several concurrent Join-rules are transient, i.e., that after the last Join-rule, nodes in a network will form a cycle. That theorem also considers the pure join model. Although in this paper we consider regular runs, since in Theorem 5.3 (un)fair-leaving is not allowed, the theorem holds for all runs. More general sequences of concurrent joining and leaving are considered in [17], where a lower bound of the rate at which nodes need to maintain the system such that it works correctly is given (again) with high probability. In the previous sense, our Corollary 5.3 corresponds to the main statement [17, Theorem 5.9] of that paper.

It is not quite clear how to compare these two approaches, but in our opinion there is benefit from both of them. One can argue that the probabilistic approach (providing lower bounds of probabilities) is useful to study robustness of protocols. On the other hand, we are able to describe sequences of actions leading to (un)stable states of Chord networks, which can be useful when one analyzes properties of systems that incorporate Chord and assume its correctness, as it is the case with non-relational database systems.

In [15] the theory of stochastic processes is used to estimate the probability that a Chord network is in a particular state. In [1, 2] Chord's stabilization algorithm is modelled using the  $\pi$ -calculus and its correctness is established by proving the equivalence of the corresponding specification and implementation. Possible departures of nodes from a network are not examined in this approach. The paper [25] presents the results of testing of Chord protocol which is based on random simulations. It reports some failures and proposes modifications in the Join and Stabilization rules. For example, an update of the successor of a node is added to the Stabilize-rule to decrease the number of steps needed to obtain a stable state. Our specification of Chord includes that modification. In [26, 27] the Alloy formal language is used to prove correctness of the pure join model. The same formalization produces several counterexamples to correctness of Chord ring-maintenance. In our approach, since we consider the regular runs only, those examples do not cause incorrectness. However, as we argue above, our results do not imply that in a more general framework some shortages cannot happen.

Finally, we note that the mentioned papers mainly consider maintaining of topological structure of Chord networks, and do not analyze handling of  $(key, value)$ -pairs presented in our Theorem 5.7 and Corollary 5.4.

## 7. CONCLUSION

In this paper we have presented an ASM-based formalization of the Chord protocol. We have proved that the proposed formalization is correct with respect to the regular runs. Up to our knowledge, it is the first comprehensive formal analysis of Chord presented in the literature which concerns both maintenance of the ring topology and data distribution. We have also indicated that if we consider all possible runs, incorrect behavior of Chord protocol could appear.



Possible direction for further work is to apply similar technique to describe other DHT protocols. For example, an interesting candidate for examination in the ASM-framework could be Synapse, a protocol for information retrieval over the interconnection of heterogeneous overlay networks defined in [18], and applied in [19].

Another challenge could be verification of the given description in one of the formal proof assistants (e.g., Coq, Isabelle/HOL). It might also produce a certified program implementation from the proof of correctness of our ASM-based specification.

**7.1. Acknowledgements.** We would like to thank Luigi Liquori for his helpful comments.

The work presented here was supported by the Serbian Ministry of Education and Science (the projects ON174026 and III44006), through Matematički Institut, and Ministarstvo znanosti, obrazovanja i športa republike Hrvatske.

## REFERENCES

- [1] R. Bakhshi, D. Gurov. *Verification of Peer-to-peer Algorithms: A Case Study*. Technical report, ICT, 2006.
- [2] R. Bakhshi, D. Gurov. *Verification of Peer-to-peer Algorithms: A Case Study*. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, Volume 181, 35–47, 2007.
- [3] G. Bella, E. Riccobene. *Formal Analysis of the Kerberos Authentication System*. In *Journal of Universal Computer Science*, vol. 3, no. 12, pages 1337–1381, 1997.
- [4] E. Börger, Y. Gurevich, D. Rosenzweig. *The Bakery Algorithm: Yet Another Specification And Verification.*, In *Specification and Validation Methods*, Oxford University Press, pages 231–243, 1995.
- [5] E. Börger, R. Stärk. *Abstract State Machines A Method for High-Level System Design and Analysis.*, Springer-Verlag, 2003.
- [6] E. Börger, A. Prinz. *Quo Vadis Abstract State Machines?* In *Journal of Universal Computer Science*, vol. 14, no. 12, pages 1921–1928, 2008.
- [7] M. Botinčan, P. Glavan, D. Runje. *Distributed Algorithms. A Case Study of the Java Memory Model*. In *Proc. of the 14th Int. ASM Workshop (ASM 2007)*, 2007.
- [8] M. Botinčan, P. Glavan, D. Runje. *Verification of causality requirements in Java memory model is undecidable PPAM*. In *Parallel Processing and Applied Mathematics 8th International Conference*, Wroclaw, Poland, September 13-16, 2009, Part II, LNCS 6068, 62 – 67, 2010.
- [9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber. *Bigtable: A distributed storage system for structured data*. In *Proceedings of the 7<sup>th</sup> Conference on Usenix Symposium on Operating Systems Design and Implementation*, Volume 7, pages 205–218, 2006.
- [10] Distributed and Mobile Systems Group Lehrstuhl für Praktische Informatik Universität Bamberg. *open-chord v. 1.0.5 implementation*, 2008.
- [11] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide*. In *Specification and Validation Methods*, Oxford University Press, pages 9–36, 1995.
- [12] Y. Gurevich. *Sequential Abstract State Machines capture Sequential Algorithms*. In *ACM Transactions on Computational Logic Volume 1, Number 1*, pages 77–111, 2000.
- [13] Y. Gurevich, J. K. Huggins. *The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions*. In *Computer Science Logic, Selected papers from CSL'95*, Lecture Notes in Computer Science 1092, pages 266–290, 1996.
- [14] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, D. Lewin. *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*. In *Proceedings of STOC'97*, pages 654–663, 1997.
- [15] S. Krishnamurthy, S. El-Ansary, E. Aurell, S. Haridi. *A statistical theory of chord under churn*. In *4th International Workshop on Peer-To-Peer Systems*, pages 93–103, 2005.
- [16] A. Lakshman, P. Malik. *Cassandra - A Decentralized Structured Storage System*. In *ACM SIGOPS Operating Systems Review*, Volume 44, Issue 2, pages 35–40, 2010.

- [17] D. Liben-Nowell, H. Balakrishnan, D. R. Karger. *Analysis of the evolution of peer-to-peer systems*. In *Proc. 21<sup>st</sup> ACM Symp. Principles of Distributed Computing (PODC)*, pages 233–242, 2002.
- [18] L. Liquori, C. Tedeschi, L. Vanni, F. Bongiovanni, V. Ciancaglini, B. Marinković. *Synapse: A Scalable Protocol for Interconnecting Heterogeneous Overlay Networks*. In *Networking 2010*, Lecture Notes in Computer Science, vol. 6091 (p. 410), pages 67–82, 2010.
- [19] B. Marinković, L. Liquori, V. Ciancaglini, Z. Ognjanović. *A Distributed Catalog for Digitized Cultural Heritage*. In *ICT Innovations 2010*, CCIS 83, pages 176 – 186, 2011.
- [20] I. Stoica, R. Morris, D. Karger, M. Kaashoek, H. Balakrishnan. *Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications*. In *ACM SIGCOMM*, pages 149–160, 2001.
- [21] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, H. Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. MIT Technical report, TR-819, 2001.
- [22] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, H. Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. In *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, 17 – 32, 2003.
- [23] R. Rodrigues, P. Druschel. *Peer-to-Peer Systems* In Communications of the ACM, Vol. 53 Issue 10, pages 72–82, October 2010
- [24] I. Taylor. *From P2P to Web Services and Grids*. Springer-Verlag, 2005.
- [25] V. Tru’o’ng. *Testing implementations of Distributed Hash Tables*. MSc thesis, IT Univesity of Göteborg, 2007.
- [26] P. Zave. *Lightweight Modeling of Network Protocols in Alloy*. <http://www2.research.att.com/~pamela/model.html>, 2010.
- [27] P. Zave. *Counterexamples to Correctness of the Chord Ring-Maintenance Protocol*. <http://www2.research.att.com/~pamela/model.html>, 2010.

## APPENDIX A. APPENDIX: CHORD RULES - LOW LEVEL DESCRIPTION

We give here a detailed specification of the module and rules introduced in Section 4.2.

```

PEER_AGENT MODULE=
// Peer tries to start/connect to the Chord network
if mode(Me) = not_connected then
  if mode_join(Me) = undef then
    if action = undef then
      // Peer has been not active and can choose to skip or to join
      choose a in Join
      action := a
    endchoose
  else
    if action = join then
      seq
        // Checking the list of known nodes of Chord network
        known := known_nodes(Me)
        if known = undef then
          // Start new Chord network
          START
        else
          // Join to existing Chord network
          Join =
            if mode_join(Me) = undef then

```

```

    JOINBEGIN
  else
    // Continuing joining process depending on received messages
    if mode_join(Me) = wait_for_successor then
      JOINGETSUCCESSOR
    else
      if mode_join(Me) = wait_for_keys then
        JOINGETKEYS
      else
        par
          // Connection was successful - start stabilization process
          if id(Me) ≠ undef then
            mode(Me) := connected
          // Connection was not successful - try again
          else
            mode(Me) := not_connected
          endif
          mode_join(Me) = undef
          action := undef
        endpar
      endif
    endif
  endif
endseq
else
  // the skip action has been chosen
  action := undef
endif
endif
// Connected node chooses what to do
else
  if mode(Me) = connected then
    // Checking if id(Me) can communicate with other nodes
    if ping(id(Me)) = true then
      par
        // Read all new messages
        READMESSAGES
        // Begin new stabilization cycle
        Stabilize =
        if mode_stabilize(Me) = undef
           $\wedge (action \neq \text{fair\_leave} \vee action \neq \text{unfair\_leave})$  then
            STABILIZEBEGIN
          else

```

```

    if mode_stabilize(Me) = wait_for_predecessor then
        STABILIZEWITHPREDECESSOR
    else
        if mode_stabilize(Me) = wait_for_keys then
            STABILIZESETKEYS
        else
            if mode_stabilize(Me) = wait_for_successor then
                STABILIZEUPDATESUCCESSOR
            else
                if mode_stabilize(Me) = wait_for_successor_keys then
                    STABILIZEUPDATEKEYS
                endif
            endif
        endif
    endif
endif
endif
endif
// Cheking connectivity of the predecessor
UPDATEPREDECESSOR
// Update value for a finger table member
UpdateFingers =
if mode_fingers(Me) = undef
     $\wedge (action \neq fair\_leave \vee action \neq unfair\_leave)$  then
        UPDATEFINGERSBEGIN
    else
        if mode_fingers(Me) = wait_for_response
            UPDATEFINGERSSET
        endif
    endif
endif
// Allow other actions beside JOIN
ExtendedJoinModel =
seq
    if action = undef then
        choose a in Action
        action := a
    endchoose
endif
par
    // Allow leaving
    LeavingActions =
    par
        FairLeave =
        if action = fair_leave  $\wedge$  mode_stabilize(Me) = undef
             $\wedge$  mode_fingers(Me) = undef then
                // Node is leaving network in a fair way
                if mode_leave(Me) = undef then

```

```

    FAIRLEAVEGETSUCCESSOR
  else
    if mode_leave(Me) = wait_for_successor then
      FAIRLEAVEUPDATESUCCESSOR
    else
      par
        FAIRLEAVEEXCHANGE
        mode(Me) := not_connected
        action := undef
      endpar
    endif
  endif
endif
if action = unfair_leave  $\wedge$  mode_stabilize(Me) = undef
 $\wedge$  mode_fingers(Me) = undef then
  par
    // Node is leaving network in a unfair way
    UNFAIRLEAVE
    mode(Me) := not_connected
    action := undef
  endpar
endif
endpar
// Allow key/value handling
KeyValueHandling =
par
  Put =
  if action = put then
    // Inserting new  $\langle key, value \rangle$  pair
    if mode_put(Me) = undef then
      seq
         $\langle key, value \rangle$  := key_value(Me)
        PUTFINDRESPONSIBLE
      endseq
    else
      par
        PUTKEYVALUE
        action := undef
      endpar
    endif
  endif
  Get =
  if action = get then
    if mode_get(Me) = undef then
      seq

```

```

        // Searching for existing value
        key := keys(Me)
        GETKEY
    endseq
else
    if mode_get(Me) = wait_for_key then
        GETVALUE
    else
        par
            GETFINISH
            action := undef
        endpar
    endif
endif
endif
endpar
endseq
endpar
else
    // Connection problems detected - reset connection
    par
        mode(Me) := not_connected
        mode_stabilize(Me) := undef
        mode_fingers(Me) := undef
        mode_leave(Me) := undef
        mode_put(Me) := undef
        mode_get(Me) := undef
        action := undef
    endpar
endif
endif
endif

START=
seq
    // Setting id(Me) according to the hash function
    id(Me) := hash(Me)
    // Set all messages to id(Me) to empty
    CLEARMESSAGES
    // Initialization of local structure
    par
        predecessor(id(Me)) := undef
        successor(id(Me)) := id(Me)
        finger(id(Me)) := []
    endpar
endseq

```

```

    next(id(Me)) := -1
    keyvalue(id(Me)) := []
  endpar
endseq

JOINBEGIN=
seq
  // Setting id(Me) according to the hash function
  id(Me) := hash(Me)
  // If connection is succesful
  if id(Me) ≠ undef then
    seq
      // Set all messages to id(Me) to empty
      CLEARMESSAGES
      // Initialize local structure
      seq
        par
          predecessor(id(Me)) := undef
          finger(id(Me)) := []
          next(id(Me)) := -1
        endpar
        // Ask for the successor
        communication(id(Me), known).add((find_successor, ⟨id(Me), id(Me)⟩))
        mode_join(Me) := wait_for_successor
      endseq
    endseq
  else
    // Connection was not succesful
    mode_join(Me) := finished
  endif
endseq

JOINGETSUCCESSOR=
// Get the message with the information on the successor
forall m ∈ Message with m = communication(sender, id(Me))
  if m = ⟨successor_found, content⟩ then
    if content = ⟨id(Me), successor⟩ then
      par
        // Set successor
        successor(id(Me)) := successor
        // Ask successor for keys
        communication(id(Me), successor(id(Me)))
          .add(⟨request_and_remove_keyvalue, undef⟩)
        mode_join(Me) := wait_for_keys
      par
        // Remove processed message

```

```

        communication(sender, id(Me)).remove(m)
    endpar
endif
endif
endforall

JOINGETKEYS=
// Get the message with keys and values
forall m ∈ Message with m = communication(sender, id(Me))
    if m = ⟨received_keyvalue, content⟩ then
        par
            // Set keys
            keyvalue(id(Me)) := content
            mode_join(Me) := finished
            // Remove processed message
            communication(sender, id(Me)).remove(m)
        endpar
    endif
endforall

FAIRLEAVEUPDATESUCCESSOR=
// Check the successor and update it if necessary
if ¬ping(successor(id(Me))) then
    seq
        communication(id(Me), id(Me)).add(⟨find_successor, ⟨id(Me) ⊕N 1, id(Me)⟩⟩)
        mode_leave(Me) := wait_for_successor
    endseq
else
    mode_leave(Me) := proceed_to_finish
endif

FAIRLEAVEGETSUCCESSOR=
// Get the message with the information on the successor
forall m ∈ Message with m = communication(sender, id(Me))
    if m = ⟨successor_found, content⟩ then
        if content = ⟨id(Me), successor⟩ then
            par
                // Set successor
                successor(id(Me)) := successor
                mode_leave(Me) := proceed_to_finish
                // Remove processed message
                communication(sender, id(Me)).remove(m)
            endpar
        endif
    endif
endforall

```



FAIRLEAVEEXCHANGE=

```

seq
  // If Me is not the last node
  if  $\neg(\text{successor}(\text{id}(\text{Me})) = \text{predecessor}(\text{id}(\text{Me})) \wedge \text{predecessor}(\text{id}(\text{Me})) = \text{id}(\text{Me}))$  then
    par
      // Send keys/values to the successor
       $\text{communication}(\text{id}(\text{Me}), \text{successor}(\text{id}(\text{Me}))).\text{add}(\langle \text{send\_keys}, \text{keyvalue}(\text{id}(\text{Me})) \rangle)$ 
      // Exchange pointers between the successor and the predecessor
       $\text{communication}(\text{id}(\text{Me}), \text{successor}(\text{id}(\text{Me}))).\text{add}(\langle \text{send\_predecessor}, \text{predecessor}(\text{id}(\text{Me})) \rangle)$ 
       $\text{communication}(\text{id}(\text{Me}), \text{predecessor}(\text{id}(\text{Me}))).\text{add}(\langle \text{send\_successor}, \text{successor}(\text{id}(\text{Me})) \rangle)$ 
      // Clean local memory
       $\text{predecessor}(\text{id}(\text{Me})) := \text{undef}$ 
       $\text{successor}(\text{id}(\text{Me})) := \text{undef}$ 
       $\text{finger}(\text{id}(\text{Me})) := []$ 
       $\text{keyvalue}(\text{id}(\text{Me})) := []$ 
    endpar
  endif
endseq

```

STABILIZEBEGIN=

```

// Check if id(Me) can communicate with other nodes
if  $\text{ping}(\text{id}(\text{Me})) = \text{true}$  then
  // Check the successor's connection
  if  $\text{ping}(\text{successor}(\text{id}(\text{Me}))) = \text{true}$  then
    par
      // Get information on the predecessor of the successor
       $\text{communication}(\text{id}(\text{Me}), \text{successor}(\text{id}(\text{Me}))).\text{add}(\langle \text{get\_predecessor}, \text{undef} \rangle)$ 
       $\text{mode\_stabilize}(\text{Me}) := \text{wait\_for\_predecessor}$ 
    endpar
  else
    par
      // Find new successor
       $\text{communication}(\text{id}(\text{Me}), \text{id}(\text{Me})).\text{add}(\langle \text{find\_successor}, \langle \text{id}(\text{Me}) \oplus_N 1, \text{id}(\text{Me}) \rangle \rangle)$ 
       $\text{mode\_stabilize}(\text{Me}) := \text{wait\_for\_successor}$ 
    endpar
  endif
endif

```

STABILIZEWITHPREDECESSOR=

```

// Get the message with the information on the predecessor of the successor
forall  $m \in \text{Message}$  with  $m = \text{communication}(\text{sender}, \text{id}(\text{Me}))$  do
  if  $m = \langle \text{received\_predecessor}, \text{content} \rangle$  then
    seq

```

```

x := content
// Remove processed message
communication(sender, id(Me)).remove(m)
// There is a new node between id(Me) and the successor
if (x ≠ undef ∧ member_of(x, id(Me), successor(id(Me)))) then
  par
    // Update the successor and take keys/values from it
    successor(id(Me)) := x
    communication(id(Me), successor(id(Me)))
      .add(⟨request_and_remove_keyvalue, undef⟩)
    mode_stabilize(Me) := wait_for_keys
  endpar
endif
// id(Me) is a new node between successor and its predecessor
if x = undef ∨ (x ≠ undef ∧ member_of(id(Me), x, successor(id(Me)))) then
  // Notify the successor to change the predecessor
  par
    communication(id(Me), successor(id(Me))).add(⟨set_predecessor, undef⟩)
    mode_stabilize(Me) := undef
  endpar
endif
endseq
endif
endforall

```

STABILIZESetKeys=

```

// Get the message with keys and values
forall m ∈ Message with m = communication(sender, id(Me))
  if m = ⟨received_keyvalue, content⟩ then
    par
      // Add keys/values to local table
      keyvalue(id(Me)).add(content)
      mode_stabile(Me) := undef
      // Remove processed message
      communication(sender, id(Me)).remove(m)
    endpar
  endif
endforall

```

STABILIZEWaitSuccessor=

```

// Get the message with the information on the successor
forall m ∈ Message with m = communication(sender, id(Me))
  if m = ⟨successor_found, content⟩ then
    if content = ⟨id(Me) ⊕N 1, successor⟩ then
      par

```

```

    // Set successor
    successor(id(Me)) := successor
    // Ask successor for keys
    communication(id(Me), successor(id(Me))
      .add(⟨request_and_remove_keyvalue, undef⟩)
    mode_stabilize(Me) := wait_for_successor_keys
    // Remove processed message
    communication(sender, id(Me)).remove(m)
  endpar
endif
endif
endforall

```

STABILIZEUPDATEKEYS=

```

// Get the message with keys and values
forall m ∈ Message with m = communication(sender, id(Me))
  if m = ⟨received_keyvalue, content⟩ then
    par
      // Add keys/values to local table
      keyvalue(id(Me)).add(content)
      mode_stabilize(Me) := undef
      // Remove processed message
      communication(sender, id(Me)).remove(m)
    endpar
  endif
endforall

```

UPDATEPREDECESSOR=

```

// Check if id(Me) can communicate with other nodes
if ping(id(Me)) = true then
  // Check the predecessor's connection
  if ping(predecessor(id(Me))) ≠ true then
    predecessor(id(Me)) := undef
  endif
endif

```

UPDATEFINGERSBEGIN=

```

// Update a finger table item
par
  next(id(Me)) := (next(id(Me)) ⊕M 1) + 1
  communication(id(Me), id(Me)).add(⟨find_successor, ⟨id(Me) ⊕N 2next(id(Me))−1, id(Me)⟩⟩)
  mode_fingers(Me) := wait_for_response
endpar

```

UPDATEFINGERSSET=

```

// Get the message with the information on the successor

```

```

forall  $m \in Message$  with  $m = communication(sender, id(Me))$ 
  if  $m = \langle successor\_found, content \rangle$  then
    if  $content = \langle id(Me) \oplus_N 2^{next(id(Me))-1}, successor \rangle$  then
      par
        // Set current finger table item
         $finger(id(Me)).listitem(next(id(Me))) := successor$ 
        // Remove processed message
         $communication(sender, id(Me)).remove(m)$ 
        // Begin update next table item
         $mode\_fingers(Me) := undef$ 
      endpar
    endif
  endif
endforall

```

```

PUTFINDRESPONSIBLE=
// Insert new  $\langle key, value \rangle$  pair into Chord network
par
  // Find the responsible node for  $hash(key)$ 
   $communication(id(Me), id(Me)).add(\langle find\_successor, \langle hash(key), id(Me) \rangle \rangle)$ 
   $mode\_put(Me) := wait\_for\_responsible$ 
endpar

```

```

PUTKEYVALUE=
// Get the message with the information on the responsible node
forall  $m \in Message$  with  $m = communication(sender, id(Me))$ 
  if  $m = \langle successor\_found, content \rangle$  then
    if  $content = \langle hash(key), successor \rangle$  then
      par
         $x := successor$ 
        if  $x \neq undef$  then
          // Send  $\langle key, value \rangle$  to the responsible node
           $communication(id(Me), x).add(\langle send\_keys, [\langle hash(key), value \rangle] \rangle)$ 
        else
          SKIP
        endif
      endpar
      // Remove processed message
       $communication(sender, id(Me)).remove(m)$ 
      // Finishing this instering operation
       $mode\_put(Me) := undef$ 
    endif
  endif
endforall

```

```

GETKEY=
// Begin search for given value
par
  communication(id(Me),id(Me)).add(⟨find_successor,⟨hash(key),id(Me)⟩⟩)
  mode_get(Me) := wait_for_key
endpar

GETVALUE=
// Get the message with the information on the responsible node
forall m ∈ Message with m = communication(sender,id(Me))
  if m = ⟨successor_found,content⟩ then
    if content = ⟨hash(key),successor⟩ then
      par
        // Ask responsible node for the value connected with the given key
        responsible := successor
        communication(id(Me),responsible).add(⟨get,hash(key)⟩)
        // Remove processed message
        communication(sender,id(Me)).remove(m)
        mode_get(Me) := wait_for_value
      endpar
    endif
  endif
endforall

GETFINISH=
// Get the message with the information on the asked value
forall m ∈ Message with m = communication(sender,id(Me))
  if m = ⟨value_found,content⟩ then
    if content = ⟨hash(key),successor⟩ then
      par
        value := content
        // Remove processed message
        communication(sender,id(Me)).remove(m)
        mode_get(Me) := undef
      endpar
    endif
  endif
endforall

SEARCH=
// Find given value in local key/value table
seq
  choose value in Value satisfying
    ⟨hash(key),value⟩ ∈ keyvalue(id(Me))
    content := value
endchoose

```

```

    communication(id(Me), sender).add(value_found, content)
endseq

CLEARMESSAGES=
// Clear messages remaining by the node which had same id
forall node ∈ Chord
    communication(node, id(Me)) := []
endforall

READMESSAGES=
// Process received messages
forall m ∈ Message with m = communication(sender, id(Me)) do
    seq
        par
            // Send proper keys/values to new predecessor
            if m = ⟨request_and_remove_keyvalue, content⟩ then
                seq
                    res := []
                    forall ⟨h, v⟩ ∈ keyvalue(id(Me)) with h ≤ sender do
                        par
                            keyvalue(id(Me)).remove(⟨h, v⟩)
                            res.add(⟨h, v⟩)
                        endpar
                    endforall
                    communication(id(Me), sender) := ⟨received_keyvalue, res⟩
                endseq
            endif
            // Add received keys/values to local table
            if m = ⟨send_keys, content⟩ then
                keyvalue(id(Me)).add(content)
            endif
            // Set given predecessor
            if m = ⟨set_predecessor, content⟩ then
                predecessor(id(Me)) := sender
            endif
            // Send information on local predecessor
            if m = ⟨get_predecessor, content⟩ then
                communication(id(Me), sender) := ⟨received_predecessor, predecessor(id(Me))⟩
            endif
            // Process query on responsible node
            if m = ⟨find_successor, content⟩ then
                FINDSUCCESSOR
            endif
            // Process search on given value in local table
            if m = ⟨get, content⟩ then

```

```

    SEARCH
  endif
endpar
// Remove processed message
communication(sender, id(Me)).remove(m)
endseq
endforall

FINDSUCCESSOR=
seq
   $\langle h, starter \rangle := content$ 
  // If  $h$  is between  $id(Me)$  and its successor responsible node is the successor
  if ping(successor(id(Me)))  $\wedge$  member_of( $h, id(Me), successor(id(Me))$ ) then
    communication(id(Me), starter).add( $\langle successor\_found, \langle h, successor(id(Me)) \rangle \rangle$ )
  else
    // Find the closest preceding node in local finger table
    seq
      index := undef
      choose  $i$  in  $\{1, \dots, M\}$  satisfying
        member_of(finger(id(Me)).listitem( $i$ ), id(Me),  $h$ )
         $\wedge \neg$ member_of(finger(id(Me)).listitem( $i + 1$ ), id(Me),  $h$ )
         $\wedge$ ping(finger(id(Me)).listitem( $i$ )) = true
        //  $h$  is between two values in local finger table
        index :=  $i$ 
    //  $h$  is bigger then all values in local finger table - the closest is the
    maximal element
    if index = undef then
      choose  $i$  in  $\{1, \dots, M\}$  satisfying
        ping(finger(id(Me)).listitem( $i$ )) = true
         $\wedge (\forall j \in \{2, \dots, M + 1 | j > i\}) ping(j) = false$ 
        index :=  $i$ 
    endif
    // Forward query to it
    communication(id(Me), finger(id(Me)).listitem(index))
    .add( $\langle find\_successor, \langle h, starter \rangle \rangle$ )
  endseq
endif
endseq

```

MATHEMATICAL INSTITUTE OF THE SERBIAN ACADEMY OF SCIENCES AND ARTS, BELGRADE,  
SERBIA

*E-mail address:* bojanm@mi.sanu.ac.rs

DEPARTMENT OF MATHEMATICS AND DESCRIPTIVE GEOMETRY, FACULTY OF MECHANICAL EN-  
GINEERING AND NAVAL ARCHITECTURE, ZAGREB, CROATIA

MATHEMATICAL INSTITUTE OF THE SERBIAN ACADEMY OF SCIENCES AND ARTS, BELGRADE,  
SERBIA